GENERIC LINKED LIST

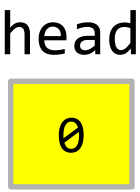
- **liked list**의 구성 원리를 단계별로 구현한다.
- 자료구조의 타입 추상화 기법을 이해한다.

1) liked list 단계별 구현
2) liked list의 library 구현
3) type에 무관한 자료구조 작성법

NULL로 끝나는 single linked list 설계

single linked list의 설계

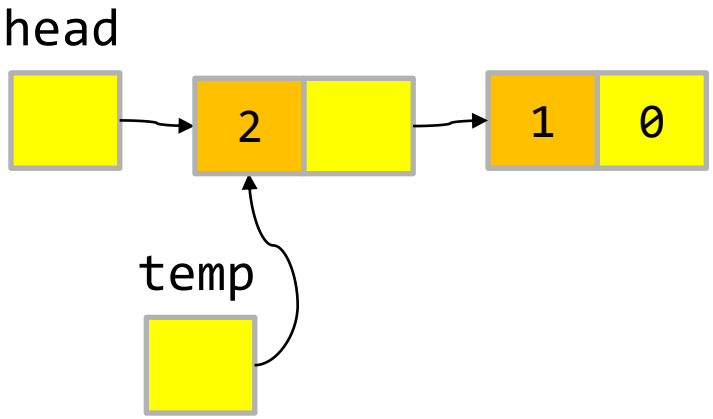
```
typedef struct _node
{
    int data;
    struct _node *next;
} NODE;

NODE *head;
```



insert_front의 구현

아래의 코드는 data가 맨 앞에 삽입되는 구조이다.

```
void insert_data( int data )
{
    NODE *temp;
    temp = malloc( sizeof(NODE) );
    temp->data = data;
    temp->next = head;
    head = temp;
}

insert_data(1);
insert_data(2);
```

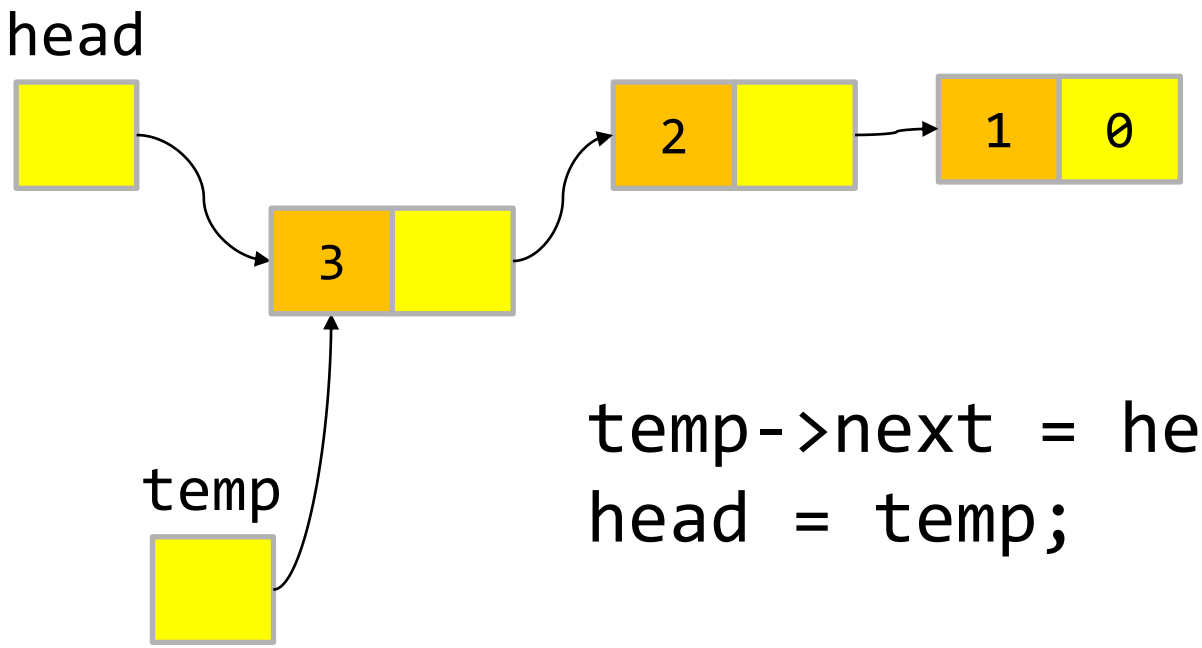


Display 구현

```
void display( void )
{
    NODE *temp;
    system("clear");
    printf("[head]");
    for(temp=head;temp != 0;temp=temp->next)
        printf("->[%d]",temp->data);
    getchar();
}
```

List 삽입 알고리즘 일반화

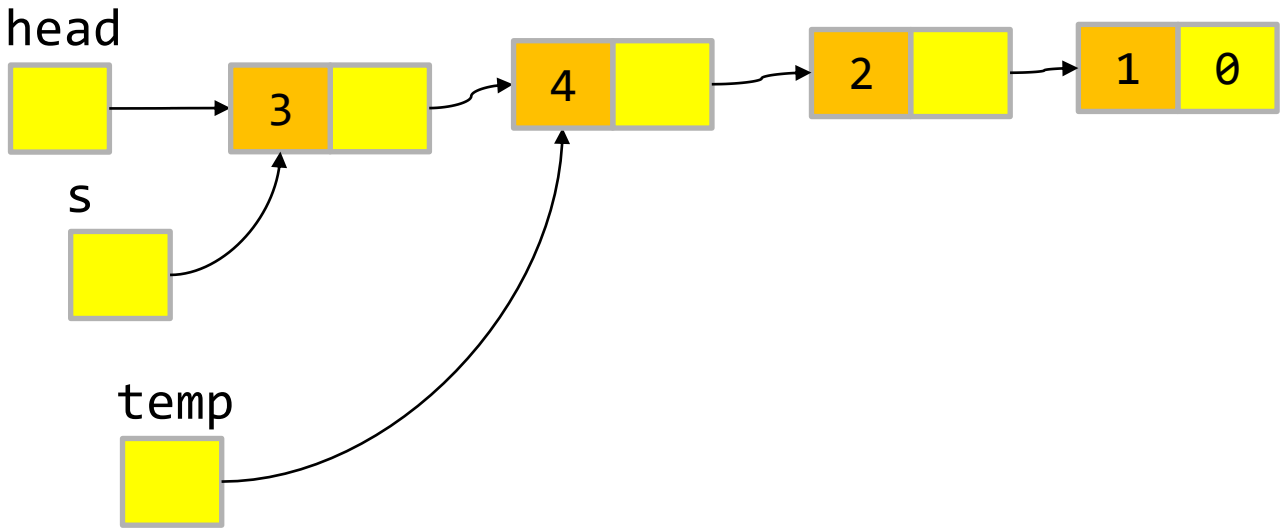
문제점 : 노드를 맨앞에 삽입할때와 그렇지 않을때의 코드가 다르다.



List 삽입 알고리즘 일반화

문제점 : 노드를 맨앞에 삽입할때와 그렇지 않을때의 코드가 다르다.

해결책 : head를 노드로 만들면 된다.

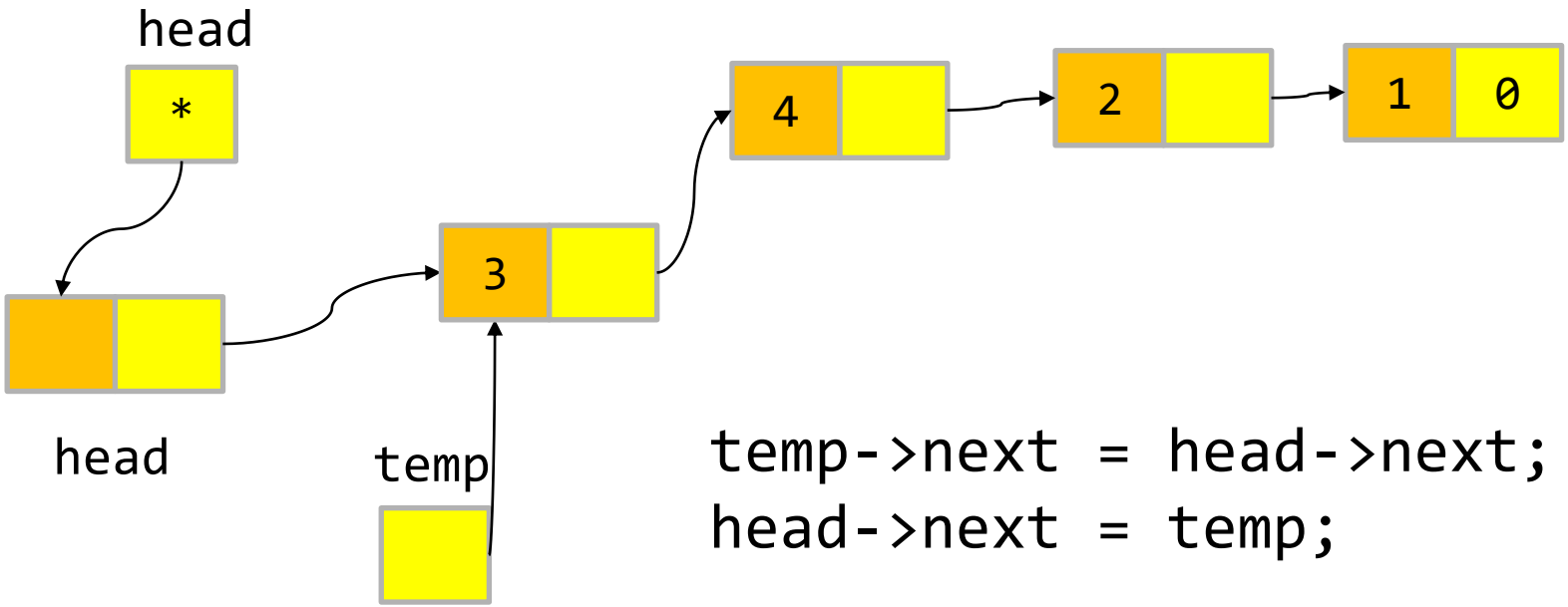



```
temp->next = s->next;
s->next = temp;
```

List 삽입 알고리즘 일반화

문제점 : 노드를 맨앞에 삽입할때와 그렇지 않을때의 코드가 다르다.

해결책 : head를 노드로 만들면 된다.




```
temp->next = head->next;
head->next = temp;
```

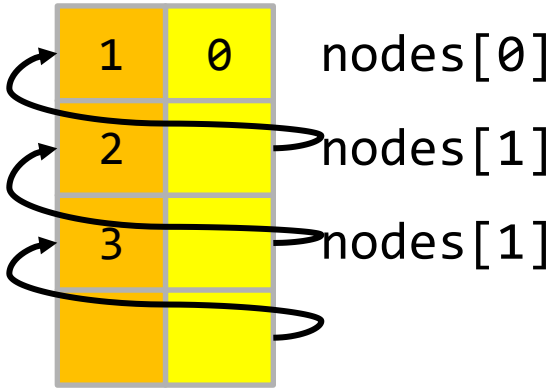
List head 의 인자처리

문제점 : list가 프로세스에 하나만 존재 가능 하다.
해결책 : head를 인자로 처리 하자.

```
void insert_data(int data , NODE *head, )
{
   NODE *temp;
   temp = malloc( sizeof(NODE) );
   temp->data = data;
   temp->next = head->next;
   head->next = temp;
}

//---------------------------------------
NODE head;
insert_data( 1, &head );
```

list의 메모리 의존성 제거

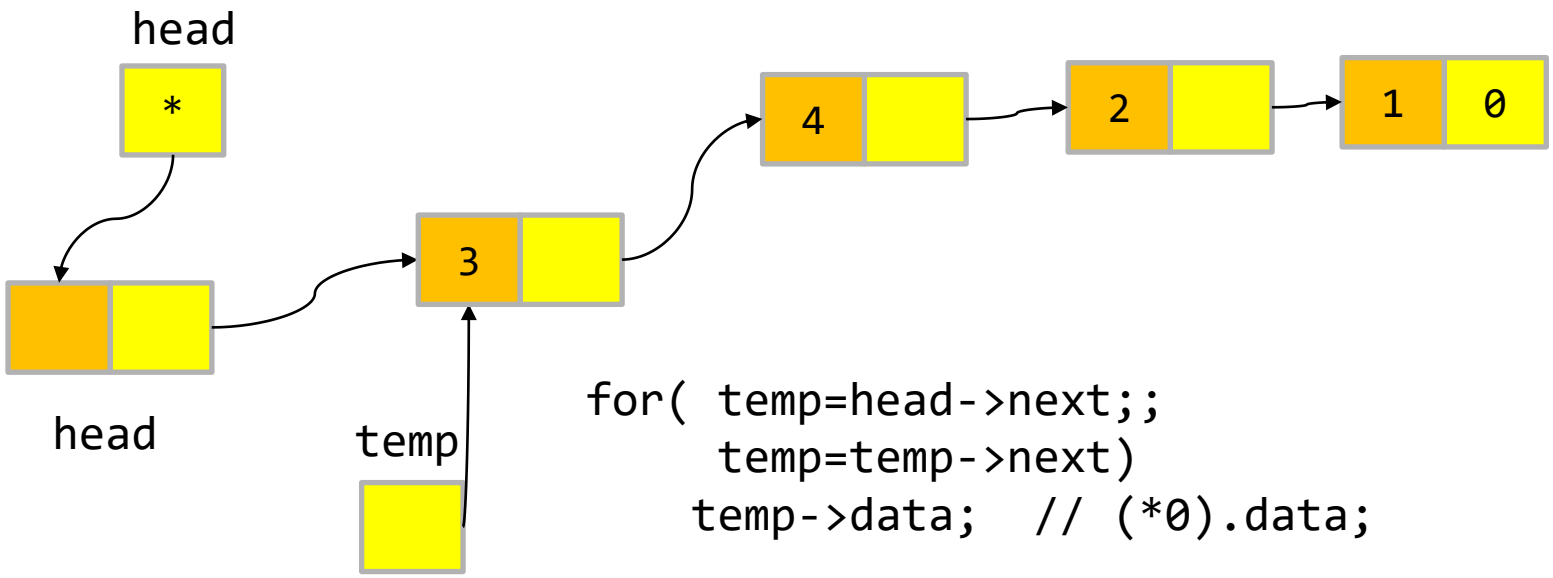
문제점 : 노드가 메모리에 의존적이다.
해결책 : 메모리의 할당/해지를 유저에 위임하자.




```
NODE head={0,};
NODE nodes[3];

void insert_data( NODE *temp, NODE *head)
{
    temp->next = head->next;
    head->next = temp;
}
```

tail node의 도입

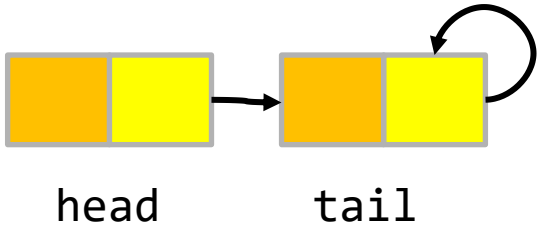
문제점 : NULL 로 끝난것이 문제
(null 참조의 위험성이 존재)
해결책 : tail로 끝나게 하자.




```
for( temp=head->next;;
     temp=temp->next)
    temp->data;  // (*0).data;
```

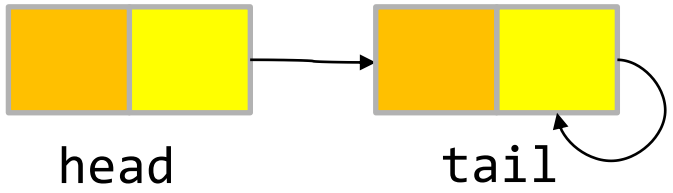
tail node의 도입

```
void display(NODE *head)
{
    NODE *temp;
    system("clear");
    printf("[head]");
    for( temp = head->next; temp != tail; temp=temp->next )
        printf("->[%d]", temp->data); // (*0).data
    printf("->[tail]\n");
    getchar();
}
NODE tail={0,&tail};
NODE head={0,&tail};
```



list의 단계별 구현 3

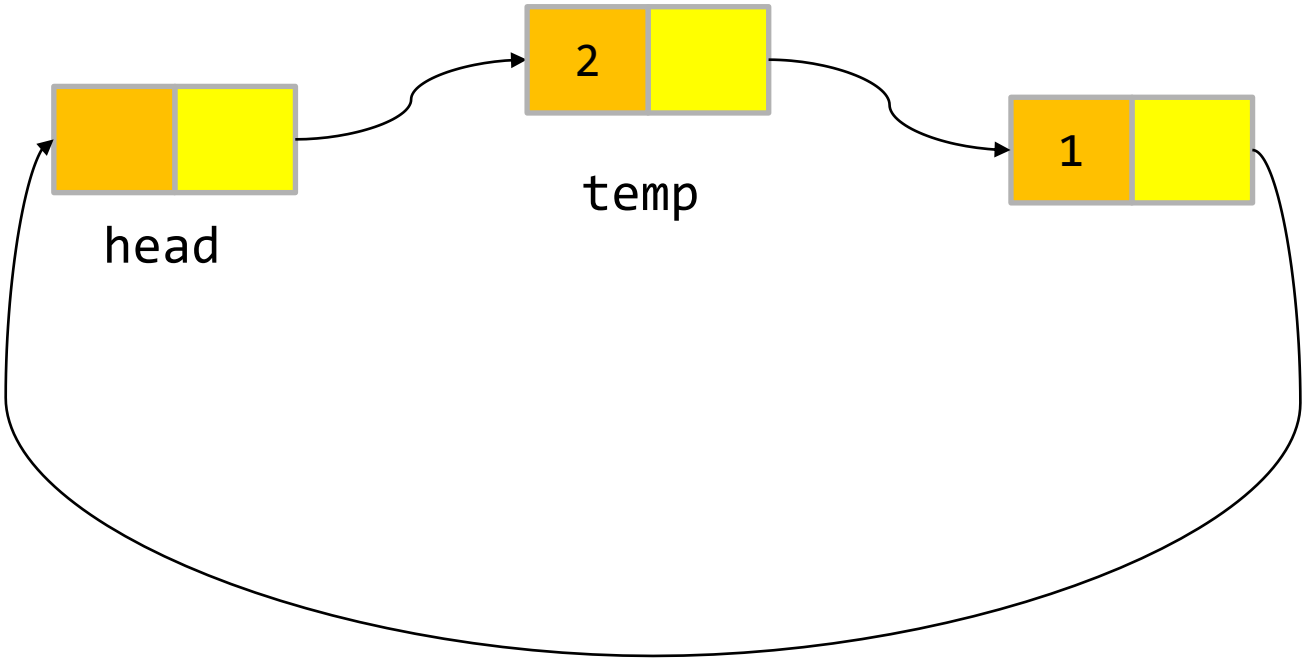
문제점 : NULL 로 끝난것이 문제(null 참조의 위험성이 존재)
해결책 : tail로 끝나게 하자.




```
for( temp=head->next; temp = &tail ;temp=temp->next)
    temp->data;  // tail.data;
```

원형 list의 도입

문제점 : 과연 tail이 필요한가?
해결책 : head가 tail의 역할을 하면된다.



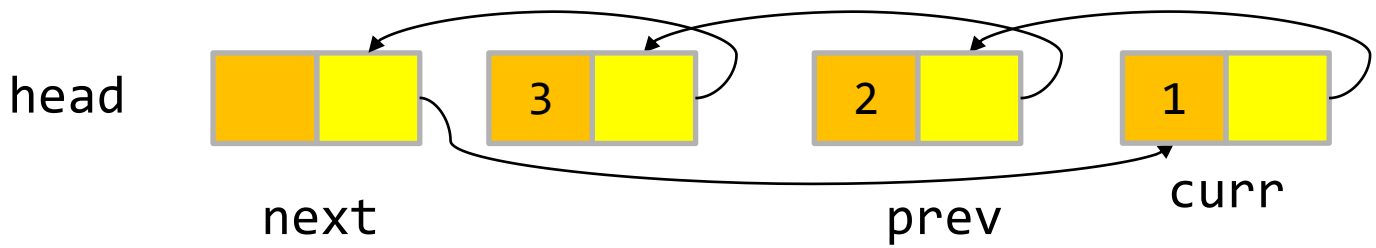

```
for( temp=head->next;temp!=head;temp=temp->next)
    temp->data;  // (*0).data;
```

원형 list의 도입

```
void display(NODE *head)
{
    NODE *temp;
    system("clear");
    printf("[head]");
    for( temp = head->next; temp!=head; temp=temp->next )
        printf("->[%d]", temp->data);
    printf("->[head]\n");
    getchar();
}
NODE head={0,&head};
```

Reverse 함수의 도입

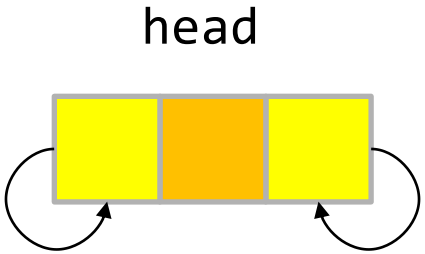
문제점 : 역순 검색을 할 수 없다.
해결책 : reverse를 구현 하면 된다.




```
void reverse( NODE *head ){
    NODE *prev=head, *curr, *next;
    curr = prev->next;
    while( curr != head )    {
         next = curr->next;
         curr->next = prev;
         prev = curr;
         curr = next;
    }
    head->next = prev;
}
```

이중연결 리스트의 도입

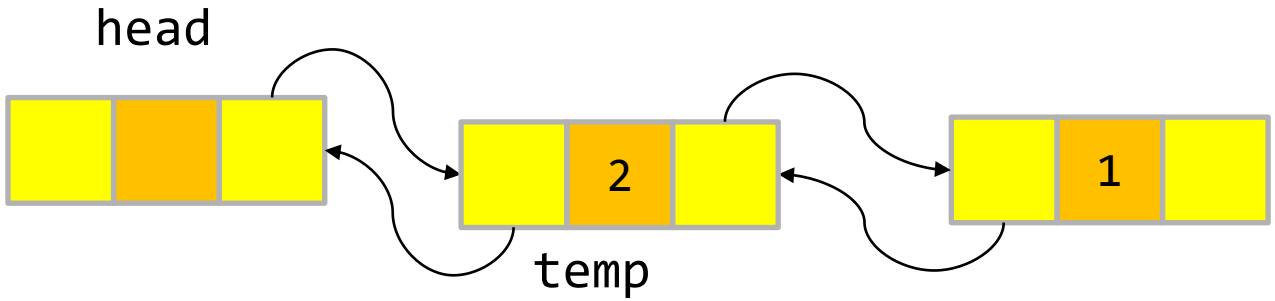
문제점 : 역순 검색을 할 수 없다.
해결책 : 이중 연결 리스트로 구현 하자.




```
typedef struct _node
{
    int data;
    struct _node *next;
    struct _node *prev;
} NODE;

NODE head = { 0, &head, &head };
```

이중연결 리스트의 insert 함수

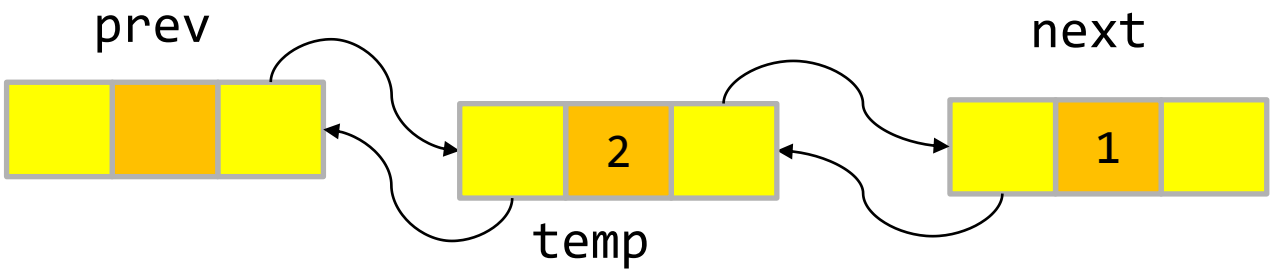
문제점 : 역순 검색을 할 수 없다.
해결책 : 이중 연결 리스트로 구현 하자.




```
void insert_data( NODE *temp, NODE *head )
{
    temp->next = head->next;
    head->next = temp;
    temp->prev = head;
    temp->next->prev = temp;
}
```

이중연결 리스트의 insert 함수의 개선

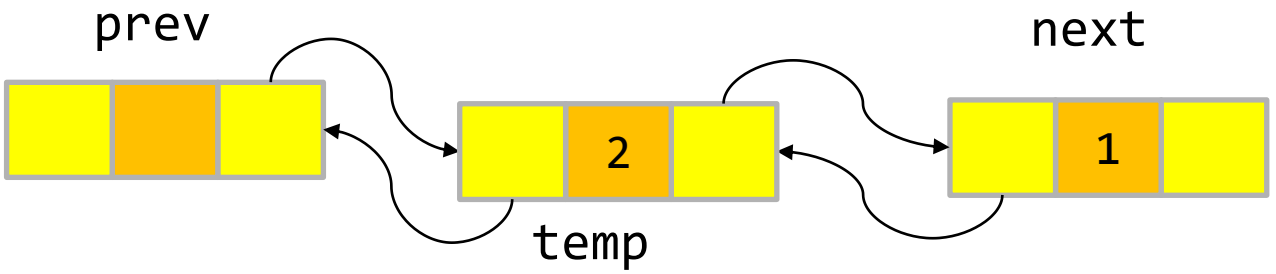
문제점 : 역순 검색을 할 수 없다.
해결책 : 이중 연결 리스트로 구현 하자.




```
void __insert_data( NODE *temp, NODE *prev, NODE *next )
{
    temp->next = next;
    prev->next = temp;
    temp->prev = prev;
    next->prev = temp;
}
void insert_front( NODE *temp, NODE *head )
{
   __insert_data( temp, head, head->next );
}
```

이중연결 리스트의 insert 함수의 개선

문제점 : 역순 검색을 할 수 없다.
해결책 : 이중 연결 리스트로 구현 하자.




```
void __insert_data(NODE *temp,NODE *prev,NODE *next)
{
    temp->next = next;
    prev->next = temp;
    temp->prev = prev;
    next->prev = temp;
}
void insert_back(NODE *temp,NODE *head)
{
    __insert_data(temp, head->prev, head );
}
```

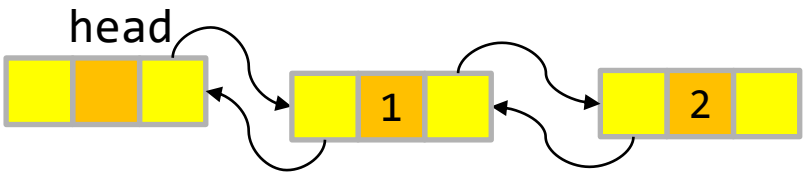
이중연결 리스트의 insert 함수의 개선

커널의 insert_front와 insert_back의 구현 예

```
void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next){
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

void list_add(struct list_head *new, struct list_head *head){
    __list_add(new, head, head->next);
}

void list_add_tail(struct list_head *new, struct list_head *head){
    __list_add(new, head->prev, head);
}
```

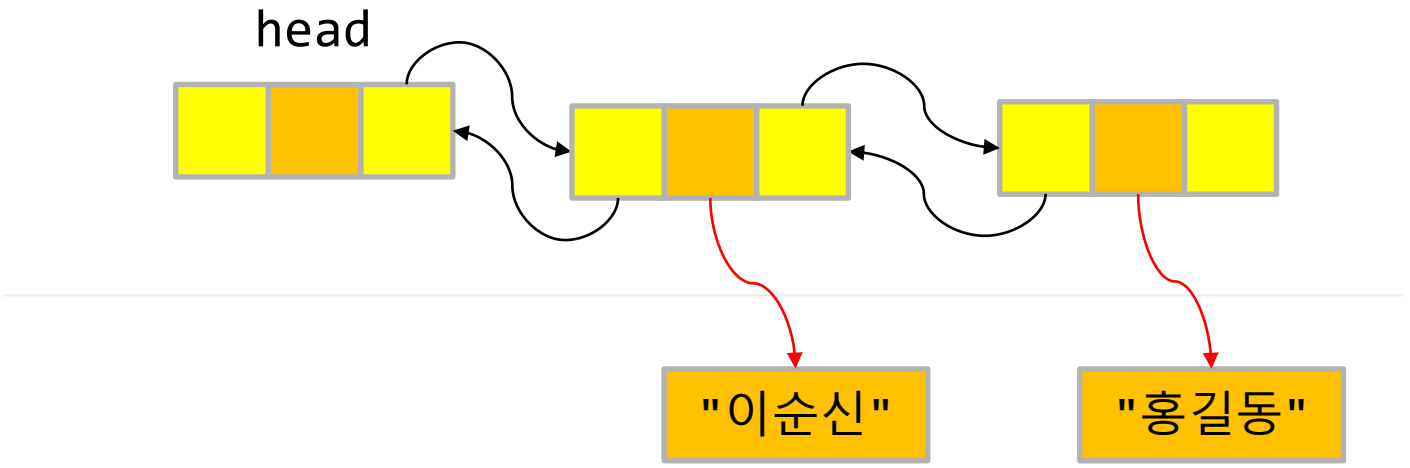
타입의 의존성 제거

```
typedef struct _node
{
    int data;
    struct _node *next;
    struct _node *prev;
} NODE;
typedef struct
{
    char name[20];
} SAWON;
void insert_back(NODE *temp,NODE *head)
{
    __insert_data(temp, head->prev, head );
}
```



타입의 의존성 제거

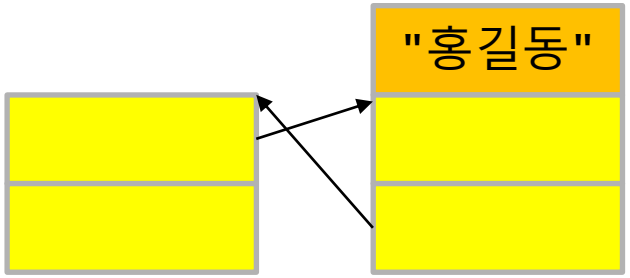
문제점 : 노드가 type에 의존적이다.
해결책 : void *를 사용하자.




```
typedef struct _node
{
    void * data;
    struct _node *next;
    struct _node *prev;
} NODE;
```

타입의 의존성 제거

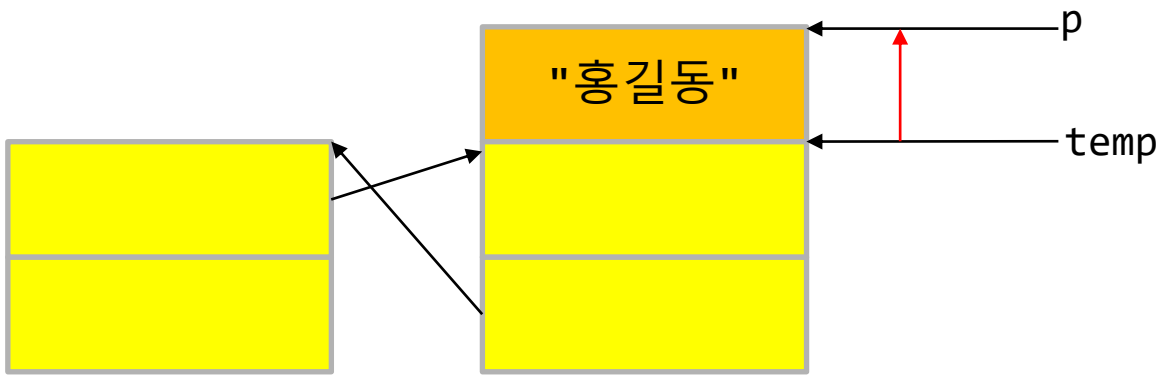
문제점 : loose coupling (느슨한 결합)
해결책 : node에서 data 포함 하지말고,
data에 node를 넣어라.




```
typedef struct _node
{
    struct _node *next;
    struct _node *prev;
} NODE;

//---------------------------------
typedef struct
{
    char name[20];
    NODE list;
} SAWON;
```

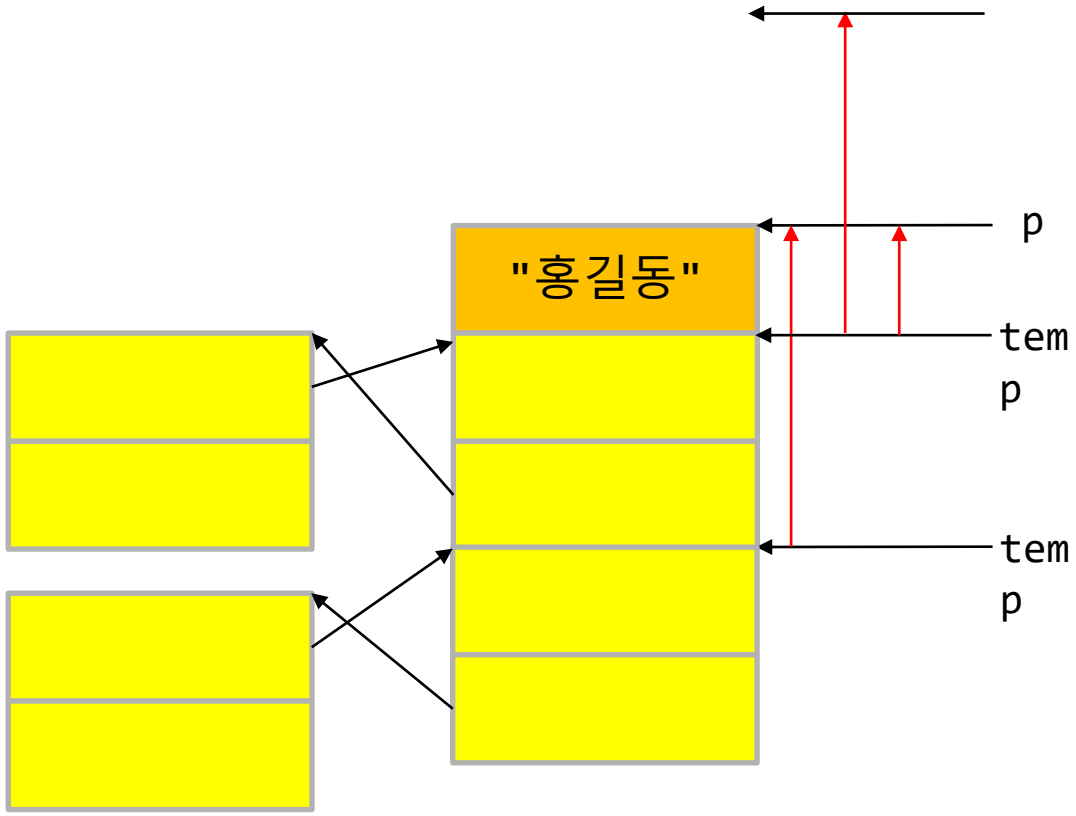
타입의 의존성 제거

문제점 : loose coupling (느슨한 결합)
해결책 : node에서 data 포함 하지말고,
data에 node를 넣어라.




```
(SAWON*)((char*)temp - (sizeof(SAWON)-sizeof(NODE)))
```

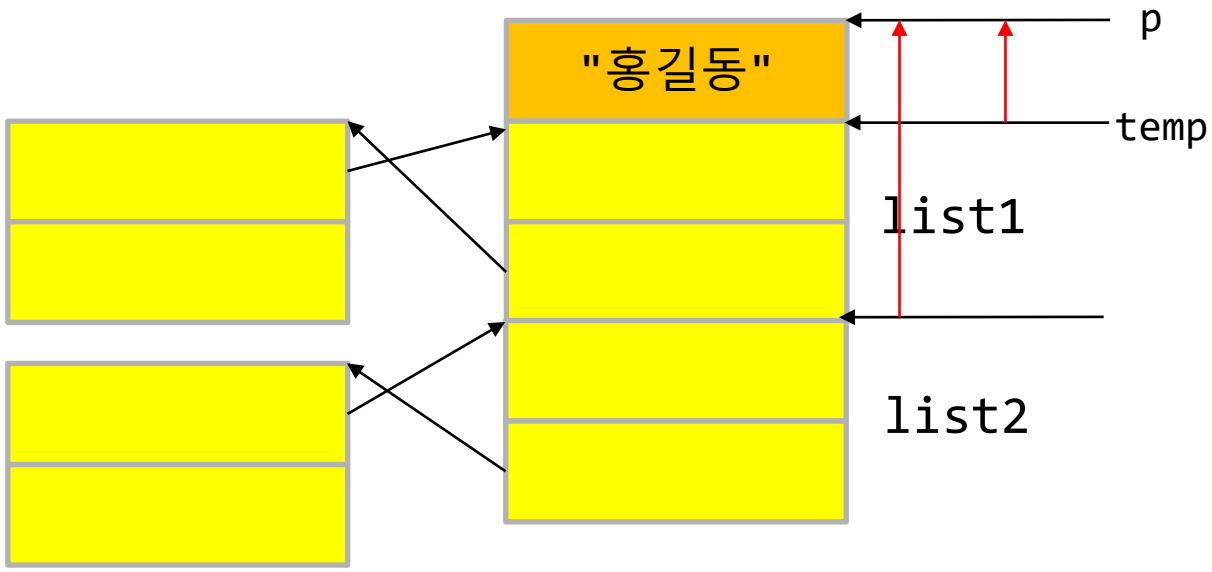
타입의 의존성 제거




```
(SAWON*)((char*)temp - (sizeof(SAWON)-sizeof(NODE)))
```

타입의 의존성 제거 (container_of 도입)

문제점 : 노드가 하나의 list에만 연결 될수 있다.
해결책 : 구조체 멤버의 offset을 이용 하라.




```
(SAWON*)((char*)temp - (unsigned long)&((SAWON*)0)->list1)
```

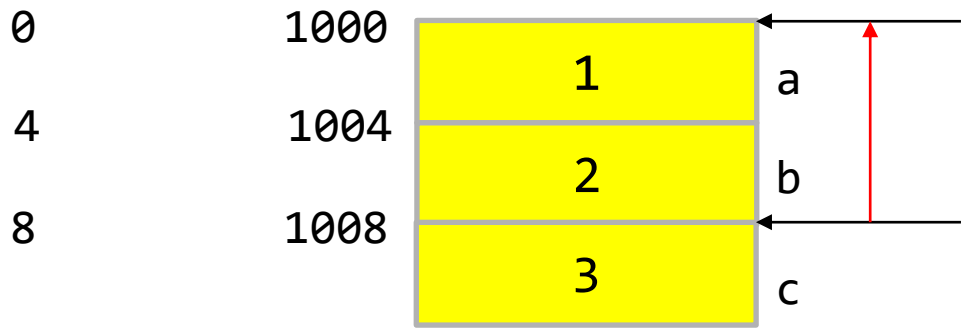
커널내 generic list의 구현예

커널내 offset_of 와 container_of의 구현 예

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)


      void display( const int *p );
      void display( const int *p )
      {
            // *p = 10;
            printf("%d\n", *p );
      }
```

구조체 멤버의 offset 계산예

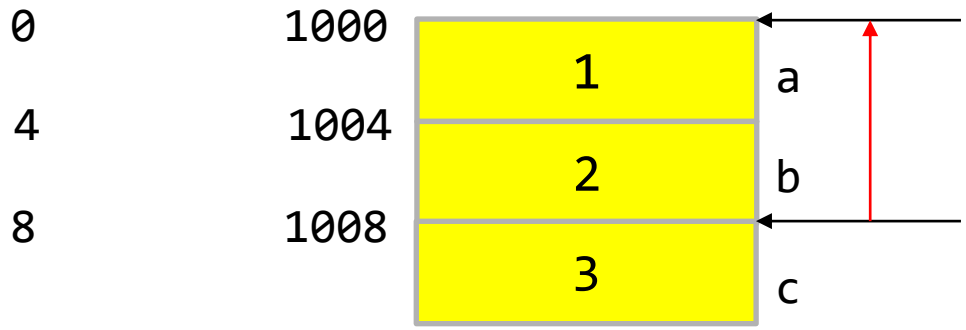
offset of 함수의 내부 구조 설명 예




```
typedef struct
{
   int a,b,c;
} AAA;

AAA aaa={1,2,3};
int *temp = &aaa.c;
AAA *p;
p = (AAA*)((char*)temp - (unsigned long)&((AAA*)0)->c);
p->a, p->b, p->c;
```

구조체 멤버의 offset 계산예

offset of 함수의 내부 구조 설명 예




```
#define  container_of(ptr, type, member)      \
(type*)((char*)ptr-(unsigned long)&((type*)0)->member)
typedef struct
{
   int a,b,c;
} AAA;
AAA aaa={1,2,3};
int *temp = &aaa.c;
AAA *p;
p = container_of( temp, AAA, c );
p->a, p->b, p->c;
```

타입의 의존성 제거 (open source 의 적용 사례)

```
struct list_head {
    struct list_head *next, *prev;
};

void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

void list_add( struct list_head *new,
               struct list_head *head )
{
    __list_add(new, head, head->next);
}
```

타입의 의존성 제거 (open source 의 적용 사례)

```
struct list_head {
    struct list_head *next, *prev;
};

void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

void list_add_tail(struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}
```

타입의 의존성 제거 (open source 의 적용 사례)

```
#define offsetof(TYPE, MEMBER)  ((size_t)&((TYPE *)0)->MEMBER)

#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

타입의 의존성 제거 (open source 의 적용 사례)

```
struct task_struct
{
    struct list_head tasks;
    struct list_head children;
    struct list_head sibling;
    char comm[16];
};

struct inode
{
    unsigned long       i_ino;
    struct list_head    i_wb_list;  /* backing dev IO list */
    struct list_head    i_lru;      /* inode LRU list */
    struct list_head    i_sb_list;
};
```

최신 container_of 매크로 분석

```
#define container_of( ptr, type, member )   \
((type*)((char*)ptr-(unsigned long)&((type*)0)->member))

#define offsetof(TYPE, MEMBER) \
      ((size_t)&((TYPE *)0)->MEMBER)

#define container_of(ptr, type, member) ({           \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```